Optimistic Lookup of Whole NFS Paths in a Single Operation

Dan Duchamp
Columbia University

Abstract

VFS lookup code examines and translates path names one component at a time, checking for special cases such as mount points and symlinks. VFS calls the NFS lookup operation as necessary. NFS employs caching to reduce the number of lookup operations that go to the server. However, when part or all of a path is not cached, NFS lookup operations go back to the server. Although NFS's caching is effective, component-by-component translation of an uncached path is inefficient, enough so that lookup is typically the operation most commonly processed by servers. We study the effect of augmenting the VFS lookup algorithm and the NFS protocol so that a client can ask a server to translate an entire path in a single operation. The preconditions for a successful request are usually but not always satisfied, so the algorithm is optimistic. This small change can deliver substantial improvements in client latency and server load.

1 Introduction

The NFS `lookup` operation frequently goes "over the wire" from client to server. For example, on the main file servers of Columbia's Computer Science department, lookups constitute approximately 31% of all NFS operations serviced. This makes `lookup` the most common operation in our environment, followed closely by `null` and `getattr`, and then by `read`.[1] Similar results are typical at other installations; lookup is the most common, or at least one of the two or three most common, NFS operation to reach the server.[2]

These results are obtained despite the presence of a cache (called "directory-name lookup cache," or DNLC) that is quite effective in mapping a *(directory-vnode, name-within-directory)* tuple to the vnode for the name. Measuring in the same environment, we found an average DNLC hit rate of 75% for name lookups in NFS file systems. Apparently, the NFS client side calls `lookup` so often that a quarter of the calls (namely, the DNLC misses) are sufficient, by themselves, to make `lookup` the most common operation at the server.

The seemingly high number of over-the-wire lookups has led us to wonder if they are all necessary and whether some steps might be taken to reduce their number. The most obvious approach is to increase the effectiveness of DNLC. We did this, in two ways:

1. The DNLC implementation of SunOS 4.1.3 will not cache a *(directory-vnode, name-within-directory)* tuple if the name is more than 15 characters long. Preliminary measurements indicated that a non-negligible fraction (12%) of DNLC misses were caused by component[3] names being longer than 15 characters. Accordingly, we increased the maximum name size to 31 characters.

   This change reduced to zero the number of DNLC misses due to over-long component names. However, the effect on the number of NFS lookups was negligible (a fraction of a percent). Investigation revealed

[1] This data was gathered by the *nfsstat* utility on eight file servers, all running SunOS version 4.1.3 and NFS version 2. The total number of NFS operations, including null, was nearly 4 million. The frequency of the other common operations was 27.5%, 22.7%, and 4.8% for `null`, `getattr`, and `read`, respectively. Across the eight servers there was considerable variance among the relative frequencies, but, in every case, `lookup`, `getattr`, and `null` were by far the most common operations.

[2] For example, the most common operations in the *nhfsstone* benchmark are, in order: lookup (34%), read (22%), write (15%), and getattr (13%).

[3] Following convention, we call the argument to VFS lookup a *path*, or *pathname*. A path consists of a sequence of *components*. The process of mapping a path or a component to a vnode we call *translation* or *resolution*.

that over-long names occurred predominantly in the local UNIX file system (UFS) rather than in remote NFS file systems.[4] This finding is obviously site-dependent and workload-dependent, so it might still be worthwhile to raise the 15-character limit, though perhaps to some number smaller than 31.

2. In the typical configuration of SunOS 4.1.3, the size of DNLC is set according to the formula "17*MAXUSERS + 90." MAXUSERS is set to 48, leading to 906 cache entries. We doubled this number, with the result of increasing the hit rate for NFS lookups by about half a percent.

The two changes together resulted in increasing the DNLC hit rate for NFS lookups by less than one percent. We conclude that most `lookup` operations that go to the server are for pathnames that have not been looked up before, or else were looked up in the "distant past."

So the simple approach of increasing DNLC size will, by itself, not substantially reduce `lookup` traffic to the server. This should not be surprising, since DNLC has been available for many years, and its performance has presumably been tuned with some care. At least in our environment, it seems that the size of DNLC has been set to beyond the point of diminishing returns.

To substantially reduce `lookup` traffic to the server requires a more efficient method for looking up "new" pathnames. In the next two sections we describe and evaluate such a method.

2 Path Lookup Algorithm

Roughly speaking, the existing lookup algorithm used at the VFS level is:

```
dir = vnode for start of path;
for (;;) {
  component = next_component(path);
  if (component is ..) {
    if (goes beyond process' root)
      return error;
    while (dir is a mount point) {
      dir = cross back over mount point;
      if (goes beyond process' root)
        return error;
    }
  }
  vnode = VOP_LOOKUP(dir, component);
  if (reached end of path)
    return vnode;
  while (vnode is mounted on)
    vnode = root of overlaid f/s;
  if (vnode is symlink)
    prepend symlink to remaining path;
  else
    dir = vnode;
}
```

[4]The DNLC module is defined at the VFS level, and is callable by any underlying file system, such as NFS or UFS.

The `VOP_LOOKUP` macro expands to call the lookup operation of the right type of underlying file system (e.g., NFS, HSFS,[5] etc.). That operation may use DNLC to reduce the number of lookup calls that go to the server; for example, both NFS and UFS do this.

This algorithm translates component names into vnodes one-by-one, testing for three major special cases at each iteration:

1. the vnode is a symlink
2. the vnode is mounted-on
3. the component is ".."

These special cases form the main reason why lookup happens one component at a time. Symlinks are hardest to handle, since they are a source of uncertainty. That is, a component cannot be known to be a symlink until the server indicates that it is, and expansion of the symlink can change the path arbitrarily. In particular, the unpredictability of the content of symlinks means that not all mount points are evident in a pathname when lookup begins. Crossing a mount point is a major operation, as it potentially changes the server to which lookup operations should be directed. Finally, references to the parent directory (i.e., ".." or "dot-dot") might also lead to crossing a mount point (in the "up" direction, as opposed to the "down" direction of the previous case).

An additional reason why the VFS lookup algorithm proceeds component-by-component is that the NFS protocol has been designed not to contain pathname syntax in the protocol because of the desirability of keeping operating system dependent detail out of the protocol specification [7]. Since NFS and UFS are the major file systems below the VFS layer, VFS algorithms have been designed to cater to their constraints.

2.1 Overview

The design of the VFS lookup algorithm is sensible, since every component must be checked for the special cases. However, the component-by-component analysis of the path is the cause of the large number of `lookup`s that go to the server. If there were no special cases, then whole paths could be looked up in a single server operation.

[5]The High Sierra file system, for CD-ROM.

In fact, the special cases seldom arise. Measuring in the same environment mentioned earlier — name lookups generated by a multi-user workload applied to eight servers over several days — we found that 97.7% of paths resolved by the VFS lookup algorithm crossed no mount points and 98.9% contained no symlinks.

Our work capitalizes on these facts. We develop a "path lookup" operation that can translate several components of a path. This operation assumes that the path includes no special cases. After a `path-lookup`, we apply some checks for the special cases. If any is found, then further `path-lookup`s may be necessary, and it is possible that some of the work performed by the first `path-lookup` may have been wasted. Hence, `path-lookup` is an "optimistic" operation. The number of `path-lookup` operations and the extent to which some of them may perform wasted work varies for each path. However, for the overwhelming majority of paths, a single `path-lookup` suffices to translate the path into a vnode. At worst, the `path-lookup` call will translate only the first component; so the ordinary `lookup` operation is the degenerate case of `path-lookup`.

Our approach is, first, to add a path-to-vnode cache at the VFS level and, second, to augment NFS as necessary to lookup whole paths whenever the path cache misses. Specifically, an additional `path-lookup` call is added to the NFS protocol; this call accepts a pathname which the server translates until the first symlink (if any) is encountered. The response contains three fields:

1. The longest symlink-free prefix of the path. The prefix may be null.
2. The file handle for the prefix.
3. The untranslated suffix of the path, with the first symlink expanded and prepended. The suffix will be null if the path contains no symlink.

Note that our additional VFS-level path cache is separate from and logically above DNLC. DNLC is used within individual file systems; the path cache is used within the VFS lookup code only. Also, the results of `path-lookup` cannot be used to fill entries in DNLC, since DNLC maps component to vnode, whereas the path cache maps path to vnode.

The `path-lookup` call should be directed only to servers that are capable of handling it. The proper approach would be to alter the MOUNT protocol so that, at mount time, the file server indicates if it can handle `path-lookup`, and, if so, which types of pathname syntax it understands. The client would then store this information in the `struct vfs` for that mount. However, to reduce the number of required protocol changes, our code assumes that every mounted file system understands the call, and tries it. If a "bad operation" RPC error occurs, or if the RPC succeeds but the server indicates that it cannot handle the syntax of the pathname, the server's inability is recorded in the `struct vfs`. Besides avoiding a change to the MOUNT protocol, this approach has the advantage of slightly easing incremental deployment. A disadvantage is that a server's limits are repeatedly re-discovered (once per mount), and automounters — which are increasingly common — tend to enormously increase the number of times that a file system is (un)mounted.

2.2 Details

This section explains how path lookup adjusts to the three special cases: symlinks, mount points, and dot-dot.

2.2.1 Symlinks

Any component of a path may be a symlink, and symlinks may expand to anything. Therefore, the servers and directories visited while translating a path are not predictable simply by inspecting the initial path.

For an example, consider the path "`./x/y/z`" illustrated in Figure 1. If `x` were a mount point, then `y/z` should be resolved in a different file system than it would be if `x` were not a mount point. The client could detect if `x` were a mount point, since the client knows its mount points. However, `x` could also be a symlink that would expand to `w`, which in turn may or may not be a mount point, leading to the same predicament.

The catch-22 is that the client cannot know which server to contact until it knows whether the path "is what it seems to be" and the client cannot know that a path is what it seems to be until its components have been looked up at the server.

To break the cycle, we optimistically assume that the path contains no special cases. Referring to the example above, the client would lookup the path `x/y/z` on the server for directory "`.`" (which is necessarily the right server for the lookup of `x`). The server responds with a partition of `x/y/z` into a symlink-free prefix and a suffix that begins with the expansion of the first symlink, if any exists.

The reason that the `path-lookup` operation translates only to the first symlink and not to the end of the path is that the optimistic assumption may be false. If the path does contain a special case, the server is probably wasting some effort translating a path that is different from the one

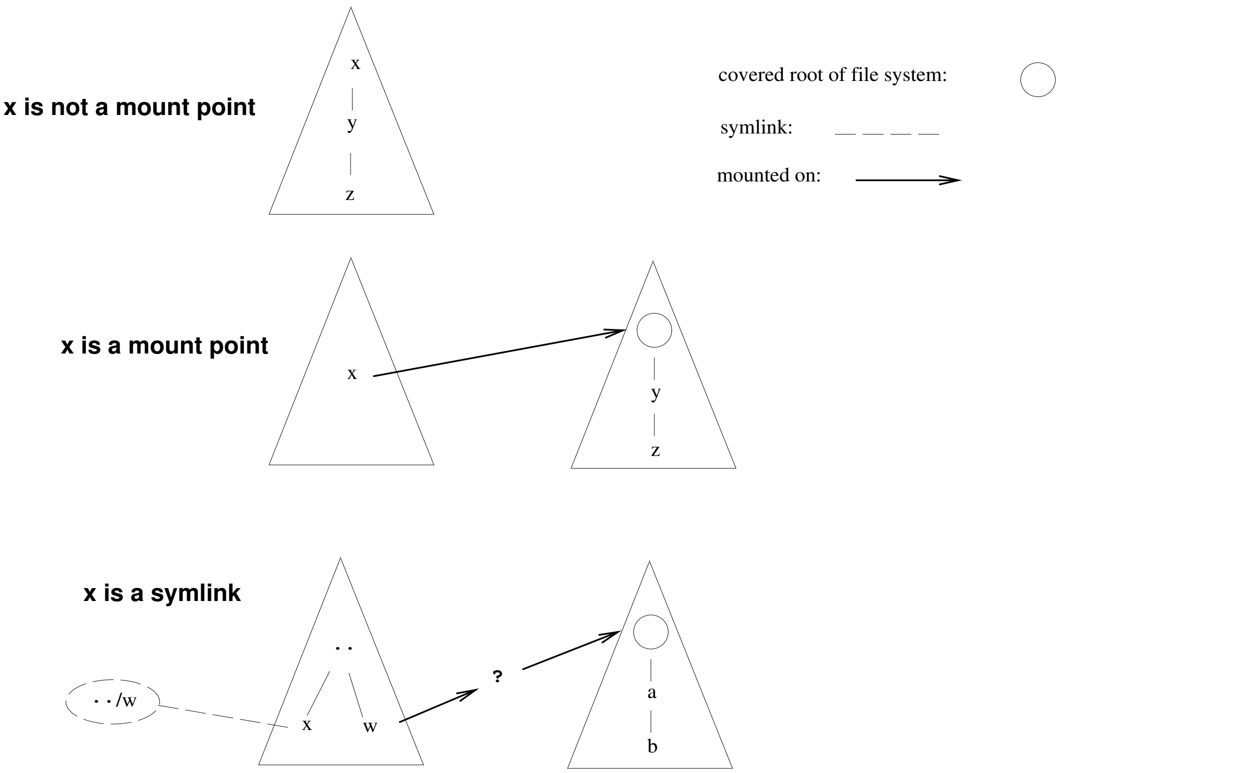

Figure 1: Uncertainty Caused By Symlinks

that should be translated. In such cases, the client must have enough information to survive the false assumption. We chose to have the server return from the `path-lookup` as soon as it encounters information that might signal that the optimistic assumption is false. Note that the server cannot return when the path crosses a mount point because the mount points that are relevant are those on the client and, practically speaking, the server cannot know the paths of the client's mount points. So the server is doing all it can by returning when it encounters a symlink. Having the server return on every symlink has essentially no effect on performance (because of the rarity of special cases) and somewhat simplifies the client (since the client need retain information for and check for only two of the three special cases).

2.2.2 Mount Points

After the `path-lookup` returns, the client will examine the symlink-free prefix for mount points. If no mount point is found, then the prefix was translated on the correct server. So the algorithm repeats by sending the suffix, if any, to the same server. If the path of some mount point is contained in the prefix, then the path lookup may have been directed to the wrong server: so the portion of the path (prefix and suffix) below the first mount point is sent to the server for the mounted file system (assuming that it understands `path-lookup`).

The reason for the mount-point check is that a server that looks up a path does so with respect to *its* name space; however, the semantics of file name translation demand that a path be translated with respect to the name space of the client. Consider the example in Figure 2. During the translation of `/usr/local/gnu/bin/emacs`, the name `gnu/bin/emacs` is translated by Server A because the client has mounted that server's file system on its name `/usr/local`. However, the client has also mounted Server B's file system on the name `/usr/local/gnu`. Therefore, the correct translation is that of `bin/emacs` with respect to Server B's file system, rather than `gnu/bin/emacs` with respect to Server A's file system. So the translation provided by Server A may be wrong.

In order to have enough information to check for mount points, the client accumulates the symlink-free prefixes returned from all `path-lookup` calls. After each call returns, the current accumulated symlink-free prefix is compared against all mount

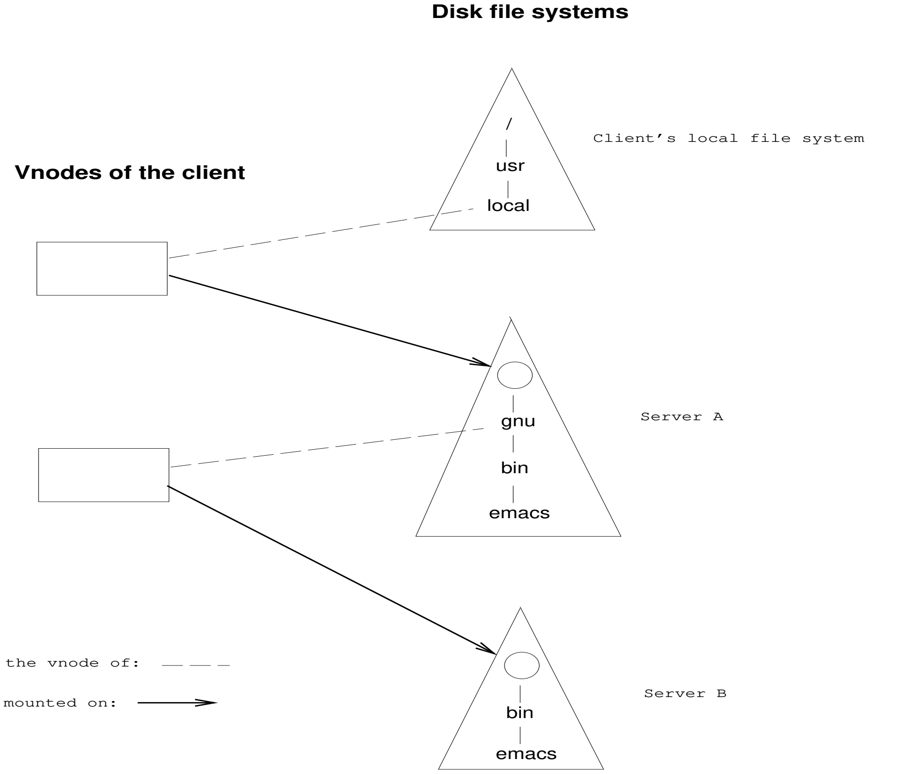

Figure 2: Why Symlink-free Path Must Be Compared Against Mount Points

points. In order to provide fast search through all mount points, we added a trie index that points to all NFS mount points. The trie stores absolute path names, as shown in Figure 3. However, the pathnames generated by a process are resolved relative to either its current root (curroot) or current working directory (cwd). This twist presents no problem: the vnodes for cwd and curroot are available, and each contains a pointer to its `struct vfs`; by definition, these structures are represented in the trie by their complete, absolute pathnames. Therefore, a pathname lies in a file system different from the one housing the starting point iff:

1. There is another mount point farther down the branch of the trie housing the `struct vfs` of the starting point.
2. The pathname is not embedded in the trie between the `struct vfs` of the starting point and the next `struct vfs`.

Our implementation platform, SunOS 4.1.3, keeps the path of all its mount points only in the file `/etc/mtab`. For three reasons, we made changes so that the name of a mount point is also kept in the associated `struct vfs`. First, for performance: the pathnames of mount points have to be accessed on every path lookup. Second, to avoid race conditions: after initiating an I/O to access `/etc/mtab` the kernel would continue; the kernel's next operation might be another that access or manipulates `/etc/mtab`. Finally, as part of earlier work [10], we had already written some code to store mount point names in `struct vfs`.

2.2.3 Dot-dot

Dot-dot must be handled with care similar to that for mount points. The reason is the same: the server will interpret dot-dot with respect to its name space, whereas the required semantics are with respect to the client's name space. Usually the two interpretations are the same. The only exception is if dot-dots in the path result in going above the root of the remote file system.[6] For example, suppose `/usr/local` is an exported file system; then the path "`/usr/local/..`" refers to `/usr` on the *client*, not the server.

Unfortunately, we thought of no simple and clean check for and adjustment to the possibility of backing up over the root of the contain-

[6] The NFS server checks for and prevents this case on every lookup.

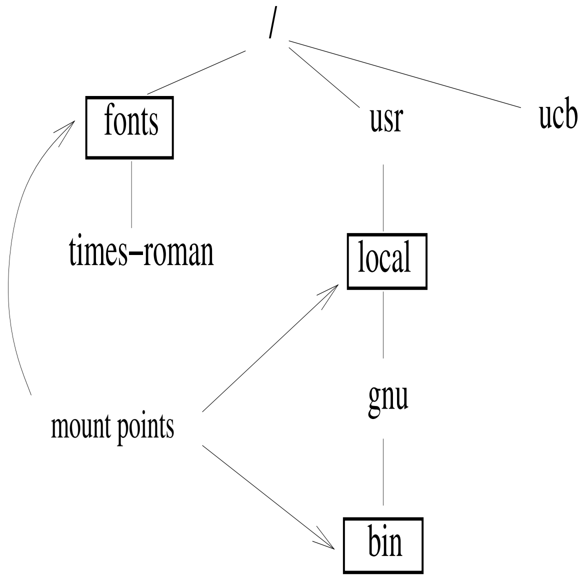

Figure 3: Trie Index for Comparing a Path Against All Mount Points

ing file system. The rub is that it is messy for the client to remember the path of every starting point (i.e., process cwd or curroot). So before each call to `path-lookup`, the client checks if the path goes above *the starting point* (i.e., cwd or curroot) of the translation. If so, the path lookup is aborted and the regular VFS lookup algorithm is used. This conservative approach means that any path that begins with dot-dot is not processed using `path-lookup`. Similarly, after the final `path-lookup` operation responds, the client checks the symlink-free prefix; if the prefix would back up over the starting point, then the path lookup algorithm is aborted.

2.2.4 Path Cache

The path cache is referenced from within the new VFS lookup algorithm and from within the NFS code for validating caches. Most of the code for the path cache was copied and adapted from DNLC.

Since NFS provides no means for a server to call back to a client, the path cache can contain outdated information.[7] Stale cache entries are removed by NFS's normal timeout-driven checking of vnode attributes and, since the cache is managed LRU, by aging the oldest entry during an insert operation. All these traits are shared with DNLC.

[7] The same is true for DNLC and, indeed, for a client-side cache of any type of information about remote NFS files.

One difference between the path cache and DNLC is that explicit deletion (such as when a file is deleted or renamed) is handled slightly differently. The delete and rename operations delete from the path cache by vnode since a vnode is a unique ID and since it would be difficult to construct a path for the target. DNLC entires can be deleted by either vnode or component name.

2.2.5 Protocol Change

The definition of the new call added to version 2 of the NFS protocol is:

```
struct pathlookupargs {
        nfspath pathname;
        int syntax;
};

struct pathlookupokres {
        nfs_fh  file;
        fattr   attributes;
        nfspath prefix;
        nfspath suffix;
};

union pathlookupres
switch (nfsstat status) {
case NFS_OK:
        pathlookupokres pathlookupres;
default:
        void;
};

pathlookupres
NFSPROC_PATH_LOOKUP(pathlookupargs) = 18;
```

A new "unintelligible syntax" error code was necessary. However, since, at the server, path lookup is simply an iterative application of the regular `lookup` operation, all the same access constraints apply.

2.2.6 Server Change

To implement the `path-lookup` operation on the server side, we stole code from other NFS operations, especially `lookup`. Essentially, the `path-lookup` implementation is that of `lookup` with two main additions:

1. Instead of translating a single component, the code iterates over components until it reaches the end, a symlink, or an error.
2. When a symlink is encountered, it is read (by calling the server-side operation to read a symlink) and prepended to the remaining untranslated path.

3 Evaluation

This algorithm is implemented in SunOS 4.1.3 and is part of the operating system regularly booted on nine SparcStations. Fewer than a thousand lines of code were added. On the server side, only a small addition was made to the module that implements NFS operations (`nfs_server.c`). Most changes were on the client side, where some modules received major changes: `vfs_lookup.c`, and a few others needed to store the path name of mount points in the `struct vfs`.

Before implementing, we studied the distribution of lengths of pathnames. The longer the pathname given to the path lookup algorithm, the greater the upside potential. Measuring in the same environment as noted earlier — eight major multi-user departmental file servers — we found considerable variation in path length distribution from machine to machine and time to time. However, on average path lengths of 4 were most common, with lengths of 3 the next most common. Paths of length 3 or 4 together accounted for over 70% of lookups. Of the remainder, most were shorter.

To measure the effect of our changes, we used a kernel build as a benchmark; all major kernel sources, libraries, and include files were on a remote file system. Within a short period of time, a kernel build opens a large number of files in a relatively small number of directories. Kernel builds provide a friendly test for path lookup: the average path length is somewhat longer (4.6) than the more comprehensive number noted above. Because of background activity, the results varied a little, but on average the build ran 8% faster with path lookup in effect. Eight percent is a substantial speedup considering that it is the effect of changing *only* the lookup operation.

Measuring the effect path lookup has on the server is at once easier and harder than measuring client latency. Measuring number of server operations is easy, but each path lookup operation can be expected to perform more work than an ordinary NFS `lookup`. We used *nfsstat* to measure the number of operations serviced and *vmstat* to record processor idle time and I/O operations. For a set of kernel build benchmarks, the number of NFS operations declined 20% and processor idle time averaged 16% higher with path lookup in effect. Apparently, processor overhead for handling NFS requests is substantial.

3.1 Violation of NFS Design Principle

As noted earlier, the addition of `path-lookup` violates the longstanding design decision to keep the NFS protocol free of path syntax.

One possible rejoinder is that, while it is true that it is desirable to keep operating system specifics out of the NFS protocol, this design decision was made several years ago; since then, NFS, though widely ported, has received almost all its use on DOS and UNIX platforms. We question whether the substantial negative impact on performance caused by component-by-component lookup is acceptable considering that the abstraction offered by omitting pathname syntax "abstracts" over effectively only two implementations.

Another, possibly better, rejoinder would be to re-design our protocol change so that it accepted and returned not paths, but rather vectors of opaque components. It would still be necessary to exchange an indication of how to interpret the components, but at least the letter, if not the spirit, of the original design principle would be preserved. We have not yet made this change to the protocol.

4 Related Work

As noted in the introduction, `lookup`, `getattr`, and `null` comprise the vast majority (over 80%) of NFS operations handled by our main servers. The high number of `null` operations is attributable to our use of the Amd automounter [5]; Amd periodically "pings" every mounted file system, and NFS `null` is the ping operation. While the high number of `null` operations can thus be dismissed as site-specific, the dominance of `getattr` and `lookup` is typical for most NFS installations — as the operation mix in *nhfsstone* indicates. So naturally there has been interest in sharply reducing the frequency of these operations.

Most such interest has focused on `getattr`, although in most experiments the motivation has not been simply to reduce the frequency of `getattr` but rather to improve the consistency guarantee provided to the client by an NFS server. One early experiment is "Spritely NFS" [6], in which a callback scheme similar to that in Sprite [4] was added to NFS with the intention of providing strict cache consistency and improving performance by eliminating the overhead of refreshing the attribute cache with `getattr`. The ideas in Spritely NFS were used in modified form in "NQNFS" [3] which is a second protocol available from the NFS implementation of 4.4BSD. NQNFS differs from Spritely NFS in that the latter requires the server to keep state indicating the cache status of files at clients; however, NQNFS borrows the "lease" idea from Gray and Cheriton [2] in order to avoid the need for servers to keep state across failures. In NQNFS, a client is allowed to cache a file for a specified period of time. The only recovery action a server must take is to wait until such time as all of its leases must have expired. Neither of these systems addresses the issue of reducing `lookups`.

More recently, the specification for version 3 of NFS [8] included, among its many changes, the requirement to return attributes as a side effect of every appropriate NFS operation. The intention is to reduce the number of separate `getattr` operations that must be invoked in order to verify attribute cache consistency.

Cache consistency and NFS version 3 are a bit far afield, but we are not aware of any attempts — besides DNLC — to reduce the cost and/or number of NFS `lookup` operations. However, the notion of looking up and caching whole or partial paths has been proposed before for new file system designs [9, 1].

In 1986 Welch and Ousterhout described "prefix tables" [9]. Prefix tables are useful in an environment where a shared global hierarchy of files is partitioned into "domains," which are spread across servers. Each client maintains a prefix table that maps file name prefixes to the servers on which the associated domains reside. Prefix table entries are hints: if a file is not where a table says it is, shorter prefixes are tried until the file is found. If a client has no prefixes at all for a file (as will be the case initially), it broadcasts the file name to all servers. Relevant prefix/server mappings are returned by all servers that have such mappings. In this way, prefix table information can be easily propagated without the requirement that any two clients have precisely the same table. And because prefix tables contain only hints that need not be correct, this method avoids creating either an availability or a consistency problem.

The prefix table idea is quite similar to our work; however, there is one major difference between the model of file system use in NFS and that in the prefix table proposal. Welch and Ousterhout describe a construct called a "remote link," which is apparently a replacement for the idea of client mounts. Distinct domains are stitched together with remote links, which are server-side mounts; that is, the client has no control over how to overlay domains on top of one another — the information is encoded in remote links in the file system, and all clients see the same arrangement of domains into a hierarchy. Implementing static mounts on the server side is a significant simplification for whole-path translation (and a significant loss of flexibility for the client). In our design, the server returns after encountering a symlink and the client must check the symlink-free path for mount points — both of these features exist only because the server cannot know the client's mount points. In Welch and Ousterhout's system, unlike in NFS, there is no complication with having the server expand a symlink and continue translating it without contacting the client. In their work, the only time a server returns a pathname to the client not completely translated is when some component of the path crosses a domain boundary: either dot-dot in the upward direction or a remote link in the downward direction. In these cases, the client must check its prefix table to learn which server to send the remainder of the path to. In summary, prefix tables is an elegant idea but one targeted for a significantly different and easier model of file system definition and use.

In [1], Cheriton and Mann describe a naming system that is scalable enough to encompass the world and general enough to name many types of objects (not just files — processes, windows, network connections, etc.) Since the features that draw most of their design attention are those that permit scaling to enormous size, it is hard to make a meaningful comparison between our work and theirs. Their system has the notion of looking up and caching whole or partial pathnames. However, they reject the notion of client mounts on the grounds that such client-specific name space management operations do not scale well and stand in the way of forming a consistent global name space. The notion of symbolic links seems absent from their design, presumably on similar grounds.

5 Summary

Measurements of NFS pathnames and `lookup` performance yield several pronounced facts:

- The hit rate of DNLC is not easily improved; nevertheless, enough `lookup` operations cannot be satisfied from DNLC so that `lookup` is the operation that most commonly goes over the wire to the server.
- Average path length is long enough so that translating a completely uncached path will often require as many as 3 or 4 `lookup` operations.
- Paths given to the VFS lookup algorithm almost never contain symlinks or cross mount points, and so can typically be translated at the server in a single operation.

Given these facts, one may question whether the elegance and relative simplicity of the VFS lookup algorithm — which translates uncached pathnames one component at a time — is sufficient compensation for its high overhead.

Indeed, the server load caused by repeated NFS `lookup` operations can be reduced by up to 16% if path lookup is used instead of component lookup. Also, path lookup can have a noticeable effect on client latency for workloads that are open-intensive.

Path lookup is not hard to implement, the only tricky aspect being that a translated path must be examined even after a "successful" translation in order to ensure that the middle of the translated path did not cross a mount point. If it did, then the pathname looked up at the server is not the path that should have been looked up — the portion of the pathname below the mount point has different meaning on the two different servers.

6 Acknowledgements

Andreas Prodromidis helped gather many of the reported statistics. Margo Seltzer suggested making the arguments/results of the `path-lookup` call be vectors of opaque components.

This work was supported in part by ONR grant number N00014-93-1-0315, and by National Science Foundation CISE Institutional Infrastructure grant number CDA-90-24735.

References

[1] D. R. Cheriton and T. P. Mann. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Trans. Computer Systems*, 7(2):147–183, May 1989.

[2] C. G. Gray and D. R. Cheriton. *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.* In *Proc. Twelfth ACM Symp. Operating Systems Principles*, pages 202–210, December 1989.

[3] R. Macklem. *Not Quite NFS, Soft Cache Consistency for NFS.* In *Proc. 1994 Winter USENIX*, pages 261–278, January 1994.

[4] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. Computer Systems*, 6(1):134–154, February 1988.

[5] J. Pendry and N. Williams. *Amd – The 4.4 BSD Automounter.* Imperial College of Science, Technology, and Medicine, London, 5.3 alpha edition, March 1991.

[6] V. Srinivasan and J. C. Mogul. *Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.* Research report 89/5, DEC Western Research Lab, May 1989.

[7] Sun Microsystems, Inc. NFS: Network File System Protocol Specification. RFC 1094, IETF Network Working Group, March 1989.

[8] Sun Microsystems, Inc. NFS: Network File System Version 3 Protocol Specification. June 25, 1993. Available from gatekeeper.dec.com: /pub/standards/nfs/nfsv3.ps.Z

[9] B. Welch and J. Ousterhout. *Prefix Tables: A Simple Mechanism for Locating Files in a Distributed System.* In *Sixth Intl. Conf. Distributed Computing Systems*, pages 184–189, May 1986.

[10] E. Zadok and D. Duchamp. *Discovery and Hot Replacement of Replicated Read-Only File Systems, with Application to Mobile Computing* In *Proc. 1993 Summer USENIX*, pages 69–85, June 1993.

Dan Duchamp is an Associate Professor of Computer Science at Columbia University. His current research interest is the various issues in mobile computing. For his initial efforts in this area, he has been named an Office of Naval Research Young Investigator.

Mail address: Computer Science Department, Columbia University, 500 West 120th Street, New York, NY 10027. Email address: `djd@cs.columbia.edu`.